A General and Parallel Platform for Mining Co-Movement Patterns over Large-scale Trajectories

ABSTRACT

1. INTRODUCTION

The prevalence of positioning devices has drastically boosted the scale and spectrum of trajectory collection to an unprecedented level. Tremendous amounts of trajectories, in the form of sequenced spatial-temporal records, are continually generated from animal telemetry chips, vehicle GPSs and wearable devices. Data analysis on large-scale trajectories benefits a wide range of applications and services, including traffic planning [1], animal analysis [2], and social recommendations [3], to name just a few.

A crucial task of data analysis on top of trajectories is to discover co-moving patterns. A *co-movement* pattern [4] refers to a group of objects traveling together for a certain period of time and the group is normally determined by spatial proximity. A pattern is prominent if the size of the group exceeds M and the length of the duration exceeds K, where M and K are parameters specified by users. Rooted from such basic definition and driven by different mining applications, there are a bunch of variants of co-movement patterns that have been developed with more advanced constraints.

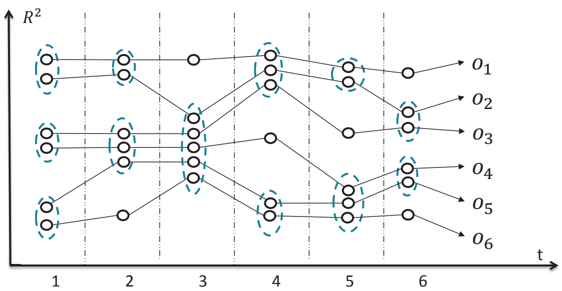
Table 1 summarizes several popular co-moving pattern s with different constraints in the attributes of clustering in spatial proximity, consecutiveness in temporal duration and computational complexity. In particular, the *flock* [5] and the *group* [6] patterns require all the objects in a group to be enclosed by a disk with radius r; whereas the *convoy* [7], the *swarm* [8] and the *platoon* [9] patterns resort to density-based spatial clustering. In the temporal dimension, the *flock* [5] and the *convoy* [7] require all the timestamps of each detected spatial group to be consecutive, which is referred to as *global consecutiveness*; whereas the *swarm* [8] does not impose any restriction. The *group* [6] and the *platoon* [9] adopt a compromised manner by allowing arbitrary gaps between the consecutive segments, which is called *local consecutiveness*. They introduce a parameter L to control the minimum length of each local consecutive segment.

	Proximity	Consecutiveness	Time Complexity
flock [10]	disk-based	global	$O(\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert(M + log(\lvert\mathbb{O}\rvert))$
convoy [7]	density-based	global	$O(\lvert\mathbb{O}\rvert^2 + \lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$
swarm [8]	density-based	-	$O(2^{\lvert\mathbb{O}\rvert}\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$
group [6]	disk-based	local	$O(\lvert\mathbb{O}\rvert^2\lvert\mathbb{T}\rvert)$
platoon [9]	density-based	local	$O(2^{\lvert\mathbb{O}\rvert}\lvert\mathbb{O}\rvert\lvert\mathbb{T}\rvert)$

Table 1: Constraints and complexity of co-movement patterns. The time complexity indicates the performance in the worst case, where $|\mathbb{O}|$ is the total number of objects and $|\mathbb{T}|$ is the number of descritized timestamps.

Figure 1 is an example to demonstrate the concepts of various co-movement patterns. The trajectory database consists of six moving objects and the temporal dimension is discretized into six snapshots. In each snapshot, we treat the clustering methods as a black-box and assume that they generate the same clusters. Objects in proximity are grouped in the dotted circles. As aforementioned, there are three parameters to determine the co-movement patterns and the default settings in this example are $M = 2$, $K = 3$ and $L = 2$. Both the *flock* and the *convoy* require the spatial clusters to last for at least K consecutive timestamps. Hence,$\{o_3, o_4\}$ and $\{o_5, o_6\}$ remains the only two candidates matching the patterns. The *swarm* relaxes the pattern matching by discarding the temporal consecutiveness constraint. Thus, it generates many more candidates than the *flock* and the *convoy*. The *group* and the *platoon* add another constraint on local consecutiveness to retain meaningful patterns. For instance, $\{o_1, o_2 : 1, 2, 4, 5\}$ is a pattern matching local consecutiveness because timestamps $\{1, 2\}$ and $\{4, 5\}$ are two segments with length no smaller than $L = 2$. The difference between the *group* and the *platoon* is that the *platoon* has an additional parameter K to specify the minimum number of snapshots for the spatial clusters. This explains why $\{o_3, o_4, o_5 : 2, 3\}$ is a *group* pattern but not a *platoon* pattern.

As can be seen, there are various co-movement patterns requested by different applications and it is cumbersome to design a tailored solution for each type. In addition, existing pattern definitions are not expressive enough and may miss interesting patterns or return noisy results. We summarize the two scenarios as *missing-pattern* anomaly and *loose-connection* anomaly. A *missing-pattern* anomaly arises due to the stringent constraints on the pattern duration. As shown in Figure 2 (a), if we set $K = 4$, neither *flock*s nor *convoy*s can be discovered. This is because o_1 is away from o_2 at timestamp 4, which is likely caused by the traffic control or the clustering inaccuracy at time 4. On



Flock(M=2, K=3)	$\{o_3, o_4$: 1,2,3$\}$ $\{o_5, o_6$: 3,4,5$\}$
Convoy(M=2, K=3)	$\{o_3, o_4$: 1,2,3$\}$ $\{o_5, o_6$: 3,4,5$\}$
Group(M=2, L=2)	$\{o_1, o_2$: 1,2,4,5$\}$ $\{o_3, o_4, o_5$: 2,3$\}$ $\{o_5, o_6$: 3,4,5$\}$
Platoon (M=2,K=3,L=2)	$\{o_1, o_2$: 1,2,4,5$\}$ $\{o_3, o_4$: 1,2,3$\}$ $\{o_4, o_5$: 2,3,5,6$\}$ $\{o_5, o_6$: 3,4,5$\}$
Swarm (M=2,K=3)	$\{o_1, o_2$: 1,2,4,5$\}$ $\{o_3, o_4$: 1,2,3$\}$ $\{o_4, o_5$: 2,3,5,6$\}$ $\{o_5, o_6$: 1,3,4,5$\}$ $\{o_2, o_3$: 3,4,6$\}$

Figure 1: Trajectories and co-movement patterns; The example consists of six trajectories across six snapshots. Objects in spatial clusters are enclosed by dotted circles. M is the minimum cluster cardinality; K denotes the minimum number of snapshots for the occurrence of a spatial cluster; and L denotes the minimum length for local consecutiveness.

the other hand, the *loose-connection* anomaly occurs due to an over-relaxed constraint on the duration. As shown in Figure 2 (b), the two objects o_1, o_2 form a *platoon* pattern $\{o_1, o_2 : 1, 2, 3, 102, 103, 104\}$. However, the consecutive segments are 98 timestamps apart, making the co-moving behavior very weak. In reality, such an anomaly is likely to be induced by the periodic movements of unrelated objects such as, vehicles stopping at the same petrol station, animals pausing at the same water source etc. It is easy to see that none of the existing co-movement patterns are able to avoid these two anomalies.

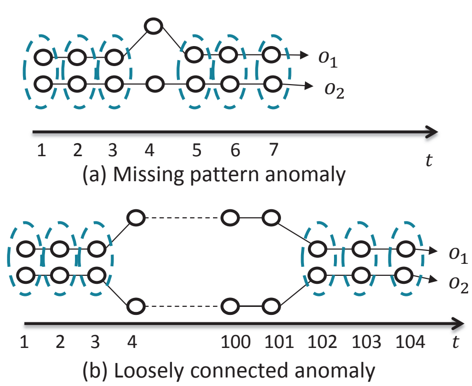
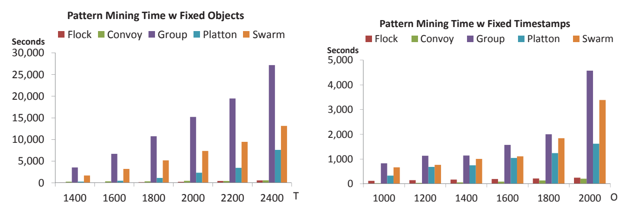


Figure 2: Two anomalies in existing patterns. (a) *Missing-pattern* anomaly in *flock* and *convoy*. When $K = 4$, none of the two patterns can be discovered. (b) *Loose-connection* anomaly in *platoon* and *swarm*. The consecutive segment of o_1 and o_2 are 98 timestamps apart, however, the pattern $\{o_1, o_2 : 1, 2, 3, 102, 103, 104\}$ is included in platoon and swarm results.

The other issue with existing methods is that they are built on top of centralized indexes that are not scalable. To the best of our knowledge, the maximum number of trajectories ever evaluated is up to hundreds of trajectories. In practice, it is rather common to collect at least hundreds of thousands of trajectories and their scalability is left unknown. We conduct a theoretical analysis on the worst-case complexity (as listed in Table 1) as well as an experimental evaluation with a real trajectory database including million-scale points(as shown in Figure 3). Results show that their performances degrade dramatically as the dataset size scales up. For instances, performance of *swarm* drops five times as the number of objects grows from *1k* to *2k*. Similarly, performance of *group* drops over seven times as the number of snapshots grows from *1.5k* to *2.4k*. It is easy to spot that none of the existing solutions are scalable to handle large-scale trajectories which may include billion-scale points.



(a) $1k$ objects (b) $1.2k$ timestamps

Figure 3: Performance measures on existing co-movement patterns. A sampled Geolife data set is used with over 2 million points.

Therefore, our primary contributions in this paper are to close these two gaps. First, we propose the *general co-movement pattern* (GCMP) which models various co-moment patterns in a unified way while avoids the two identified anomalies. In GCMP, we gain a fine-grained control over pattern duration by introducing the gap parameter G, which enforces the gap between timestamps to be no larger than G. The adoption of G seamlessly integrates with other pattern parameters, which preserves the expressiveness while alleviates both the *missing-pattern* and the *loose connection* anomalies.

Second, we propose a parallel solution on modern MapReduce platforms for scalable pattern mining. The major challenge in designing MapReduce-based algorithms is to make proper partitions of input data. In GCMP mining, we enforce both the *soundness* and the *completeness* of the partitions. Such properties ensure that neither false-patterns nor miss-patterns are possible in our solution. To meet such partition requirements as well as to keep the shuffle amount to a minimum, we first design a naive *Temporal Replication and Mining* (TRM) approach, which partitions trajectories into groups of consecutive snapshots. Then, we design a line-sweep method for mining GCMP from each partition. We prove that in TRM, the partition is complete and sound when a snapshot is replicated $O(|T|)$ times. Then, we design a novel *Star Partition and Mining* (SRM) approach which significantly reduces the data shuffled as compare to TRM. In SRM, we design a conceptual connection graph based on proximity among objects. We adapt a *star partition* which cut the graph by replicating vertices. Afterwards, we design an Apriori-like method to mine GCMP in each partition. We prove the correctness of SPM and show that total data been replicated is $O(|\mathbb{O}|)$. Despite the simpleness of star partition, we theoretically prove its optimality. Furthermore, we utilize *temporal monotonicity* to further reduce the shuffling and mining cost in SPM.

We conduct a set of extensive experiments on XXX datasets with million-scale trajectories. The results show that XXX.

The rest of our paper is organized as follows: Section 2 summarizes the relevant literature on trajectory pattern mining; Section 3 forms the definition of the general co-movement pattern mining; Section 4 presents our parallel architecture; The solution of mining the general co-movement pattern mining is presented in Section 5 and Section 6. Section 7 discuss various optimization techniques to boost the system performance; Section 8 conducts extensive experiments to showcase the usefulness and efficiency of our system and finally Section 9 concludes our paper.

2. RELATED WORKS

The *co-movement patterns* in literature consist of five members, namely *group* [6], *flock* [10], *convoy* [7], *swarm* [8] and *platoon* [9]. We have demonstrated the semantics of these patterns in Table 1 and Figure 1. In this section, we focus on comparing the techniques used in these works. For more trajectory patterns other than *co-movement patterns*, interested readers may move to [11] for a comprehensive survey.

2.1 Flock and Convoy

The difference between *flock* and *convoy* lies in the object clustering methods. In *flock* objects are clustered based on their distance. Specifically, the objects in the same cluster needs to have a pair-wised distance less than *min_dist*. This essentially requires the objects to be within a disk-region of delimiter less than *min_dist*. In contrast, *convoy* cluster the objects using density-based clustering [12]. Technically, *flock* utilizes a m^{th}-order Voronoi diagram [13] to detect whether a subset of object with size greater than m stays in a disk-region. *Convoy* employs a trajectory simplification [14] technique to boost pairwise distance computations in the density-based clustering. After clustering, both *flock* and *convoy* use a line-sweep method to scan each snapshots. During the scan, the object group appears in consecutive timestamps is detected. Meanwhile, the object groups that do not match the consecutive constraint are pruned. However, such a method faces high complexity issues when supporting other patterns. For instance, in *swarm*, the candidate set during the line-sweep grows exponentially, and many candidates can only be pruned after the entire snapshots are scanned.

2.2 Group, Swarm and Platoon

Different from *flock* and *convoy*, all the *group*,*swarm* and *platoon* patterns have more constraints on the pattern duration. Therefore, their techniques of mining are of the same skeleton. The main idea of mining is to grow object set from an empty set in a depth-first manner. During the growth, various pruning techniques are provided to prune unnecessary branches. *Group* pattern uses the Apriori property among patterns to facilitate the pruning. *Swarm* adapts two more pruning rules called backward pruning and forward pruning. *Platoon* further adapts a prefix table structure to guide the depth-first search. As shown by Li et.al. [9], *platoon* outperforms other two methods in efficiency. However, the three patterns are not able to directly discover the general co-movement pattern. Furthermore, their pruning rules heavily rely on the depth-first search nature, which lost its efficiency in the parallel scenario.

THESE WORKS ARE MOST RELATED TO OUR PROBLEMS, SO I REMOVED OTHER RELATED WORKS FOR NOW.

3. DEFINITIONS

Let $\mathbb{O} = \{o_1, o_2, ..., o_n\}$ be the set of objects and $\mathbb{T} = \{1, 2, ..., m\}$ be the descritized temporal dimension. A time sequence T is defined as a subset of $\mathbb{T}$, i.e., $T \subseteq \mathbb{T}$, and we use $|T|$ to denote sequence length. Let T_i be i-th entry in T and we say T is consecutive if $\forall 1 \leq i \leq |T|-1$, $T_{i+1} = T_i+1$. It is obvious that any time sequence T can be decomposed into consecutive segments and we say T is *L-consecutive* [9] if the length of all the consecutive segments is no smaller than L.

As illustrate in Figure 2, patterns adapting the notion of L-consecutiveness (e.g., *platoon* and *group*) still suffer from *loose connection* problem. To avoid such an anomaly without losing pattern generality, we introduce a parameter G to control the gaps between timestamps in a pattern. Formally, a G-connected time sequence is defined as follows:

Definition 1 (G-connected)**.** *A time sequence T is G-connected if the gap between any of its neighboring timestamps is no greater than G. That is $\forall T_i, T_{i+1} \in T, T_{i+1} - T_i \leq G$.*

We take $T = \{1, 2, 3, 5, 6\}$ as an example, which can be decomposed into two consecutive segments $\{1, 2, 3\}$ and $\{5, 6\}$. T is not 3-consecutive since the length $\{5, 6\}$ is 2. Thus, it is safe to say either T is 1-consecutive or 2-consecutive. On the other hand, T is 2-connected since the maximum gap between its neighboring time stamps is $5 - 3 = 2$. It is worth noting that T is not 1-connected because the gap between T_3 and T_4 is 2 (i.e., 5-3=2).

Given a trajectory database descritized into snapshots, we can conduct a clustering method, either disk-based or density-based, to identify groups with spatial proximity. Let T be the set of timestamps in which a group of objects O are clustered. We are ready to define a more general co-movement pattern:

Definition 2 (General Co-Movement Pattern)**.** *A general co-movement pattern finds a set of objects O satisfying the following five constraints: 1) closeness: the objects in O belong to the same cluster in the timestamps of T; 2) significance: $|O| \geq M$; 3) duration: $|T| \geq K$; 4) consecutiveness: T is L-consecutive; and 5) connection: T is G-connected.*

There are four parameters in our general co-movement pattern, including object constraint M and temporal constraints K, L, G. By customizing these parameters, our pattern can express other patterns proposed in previous literature, as illustrated in Table 2. In particular, by setting $G = |T|$, we achieve the *platoon* pattern. By setting $G = |T|, L = 1$, we achieve the *swarm* pattern. By setting $G = |T|$, $M = 2$, $K = 1$, we gain the *group* pattern. Finally by setting $G = 1$, we achieve the *convoy* and *flock* pattern. In addition to the flexibility of representing other existing patterns, our GCMP is able to avoid the *loose connection* anomaly by tuning the parameter G. It is notable that GCMP cannot be modeled by existing patterns. AS MENTIONED IN WECHAT, POLISH THIS PART.

It is also observable that the number of patterns in GCMP could be exponential under some parameter settings (i.e.,

Pattern	M	K	L	G	Clustering
Group	2	1	2	$\lvert T\rvert$	Disk-based
Flock	$\cdot$	$\cdot$	K	1	Disk-based
Convoy	$\cdot$	$\cdot$	K	1	Density-based
Swarm	$\cdot$	$\cdot$	1	$\lvert T\rvert$	Density-based
Platoon	$\cdot$	$\cdot$	$\cdot$	$\lvert T\rvert$	Density-based

Table 2: Expressing other patterns using GCMP. $\cdot$ indicate a user specified value. M represents the object *size* constraints. K represents *duration* constraint. L represents *consecutiveness* constraint. G represents the *connection* constraints.

when expressing *swarm*). In particular, given a parameter M, if a pattern P is valid, then any subset of P with size M is also a valid pattern. This results in additional $\Sigma_{M \geq i \geq |P.O|} \binom{|P.O|}{i}$ patterns, which is clearly overwhelming and redundant. For all these patterns, output P is sufficient. Therefore, we define the *Closed General Co-Movement Pattern* as follows:

Definition 3 (Closed General Co-Movement Pattern). *A general co-moving pattern $P = \langle O : T \rangle$ is closed if and only if there does not exist another general co-moving pattern P' s.t. $P.O \subseteq P'.O$.*

For example, let $M = 2, K = 2, , L = 1, G = 1$. In Figure 1, the pattern $P_1 = \{o_3, o_4 : 1, 2, 3\}$ is not a closed pattern. This is because $P_2 = \{o_3, o_4, o_5 : 2, 3\}$ is a closed pattern since $P_2.O \supset P_1.O$. The closed pattern avoids outputting duplicate information, thus making the result patterns more compact.

Our definition of GCMP is free from clustering method. Users are able to supply different clustering methods to facilitate different application needs. We currently expose both disk-region based clustering and DBSCAN as options to the user.

In summary, the goal of this paper is to present a parallel solution for discovering closed GCMP from large-scale trajectory data.

Before we move on to the algorithmic part, we list the notations that are used in the following sections.

Symbols	Meanings
Tr_i	Trajectory of object i
S_t	Snapshot of objects at time t
$\mathbb{O}$	Set of objects
M	Object size constraint
K	Duration constraint
L	Consecutiveness constraint
G	Connection constraint
T	Time sequence
$C_t(o)$	the cluster of object o at time t
Sr_i	The star structure of object i

Table 3: Notions that will be used

4. OVERVIEW OF MINING GCMP IN PARALLEL

We adapt the MapReduce paradigm for designing a parallel solution of mining GCMP. In this section, we briefly describe the preliminaries on MapReduce and then describe the overview of our framework in mining GCMP.

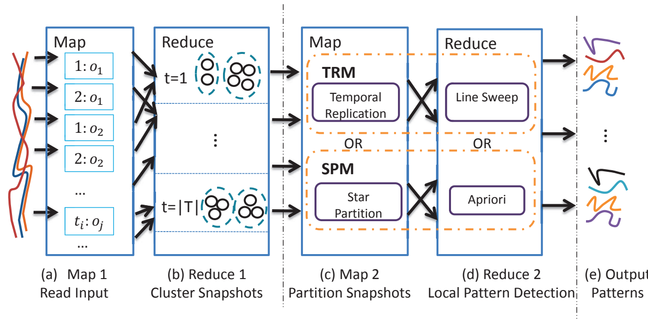


Figure 4: System flow of mining GCMP. (a)(b) correspond to the first MR job which compute the clusters at each snapshot; (c)(d) correspond to the second MR jobs which mines GCMP in parrallel.

4.1 Preliminary on MapReduce

MapReduce (MR) was formally proposed by Dean et.al. [15] and has subsequently implemented by many open source systems. Those systems provide handy APIs with fault tolerances and are popularly used as large-scale data processing platforms. In simple words, there are two conceptual types of computing nodes in MR, namely the *mappers* and the *reducers*. The execution of a MR algorithm consists of three major steps: First, input data are partitioned and read by a *map* function on each mapper. Then, mappers emit key-value pairs which are *shuffled* over the network to reducers. Lastly, reducers process the received data using a *reduce* function.

Despite the simpleness of the paradigm, there are two concerns raised in designing MR algorithms First, since reducers are required to be independent, partitioning data to fit the independence could be challenging. Second, since the *shuffle* step requires network access, the data been shuffled should be minimized. We take these concerns in consideration when designing our solutions for mining GCMP.

4.2 MapReduce Processing for Mining GCMP

Our GCMP mining process consists of two MR jobs as illustrated in Figure 4. The first MR job is to cluster objects at each snapshot (i.e., $\forall t, o$, compute $C_t(o)$). As shown in Figures 4(a)-(b), in map phase, trajectories are firstly reorganized by timestamps and object locations at the same timestamps form a snapshot. In reduce phase, clustering of objects at each snapshot is processed independently. The second MR job is to mine GCMPs from clusters in each snapshot. We design and compare two MR algorithms (i.e., TRM and SPM)for GCMP mining. Both the two algorithms would partition snapshots in map phase (as in Figure 4(c)) and mine GCMPs from each partition in reduce phase(as in Figure 4(d)).

Although we need two MR jobs to complete the GCMP mining task, it is easy to pipeline the two jobs to exploit data locality. Specifically, the reducer output at step (b) can be directly reused as the input to the mappers at step (c). Therefore we do not need to transfer data between the two jobs. Modern MR platforms, especially Spark, have already supported such a kind of pipeline.

We note that the first MR job is easy to design since each reducer only needs one snapshot for clustering. In contrast, it is challenging to design the second job. This is because valid patterns may spray across multiple snapshots or contain different object sets, where inappropriate partitioning of snapshots may fail to discover certain valid patterns. Formally, a valid partition strategy needs to meet the following requirements: (a) the resulted partitions need to preserve enough information so that real patterns can be discovered in the reduce phase. (b) the resulted partitions need to ensure that the patterns discovered in the reduce phase are valid patterns so that no further verification is required. We formalize these two properties as *completeness* and *soundness* as follows:

Definition 4 (Completeness and Soundness)**.** *Let a partition method $\mathbb{P}$ partitions original trajectories Tr into multiple parts, $Par_1, ..., Par_m$. $\mathbb{P}$ is complete if for every pattern P that is valid in Tr, $\exists Par_i$ s.t. P is valid in Par_i. $\mathbb{P}$ is sound if for all patterns that are valid in any Par_i, they are also valid in TR.*

The completeness ensures that no true patterns are missed out. The soundness ensures that no false patterns are reported. If a partition method is both sound and complete, then it can be used in the second MR job to facilitate GCMP mining.

Apparently, replicating the entire trajectories to each partition meets the *soundness* and *completeness* requirements. However, it burdens the network shuffle and limits the parallelism. Our objective is thus to design a complete and sound partition method that minimize the network shuffles. In the following sections, we describe a naive *temporal-based* partition-and-mining method called *Temporal Replication and Mining*(TRM) towards a parallel solution of GCMP mining. Then, we present a novel *object-based* partition-and-mining method called *Star Partition and Mining* (SPM) which resolves the deficiencies of TRM method.

5. TEMPORAL REPLICATION AND MINING

The idea of *Temporal Replication and Mining* (TRM) is to group temporally contiguous snapshots together, s.t. patterns can be mined from each group of snapshots. In order to achieve the *completeness* and *soundness* during partitioning, we allow replication of snapshots among groups. The TRM is outlined in Algorithm 1.

Algorithm 1 Temporal Replication and Mining

Require: list of $\langle t, S_t \rangle$ pairs
1: —Map Phase—
2: **for all** $\langle t, S_t \rangle$ **do**
3:     **for all** $i \in 1...(K-1)*G+K$ **do**
4:         emit a $\langle t-i, S_t \rangle$ pair
5:     **end for**
6: **end for**
7: —Partition and Shuffle Phase—
8: **for all** $\langle t, S \rangle$ pair **do**
9:     group-by t, emit a $\langle t, Par_t \rangle$,
10:     where $Par_t = \{S_t, S_{t+1}, ..S_{t+(\lceil \frac{K}{L} \rceil -1)*G+2K}\}$
11: **end for**
12: —Reduce Phase—
13: **for all** $\langle t, Par_t \rangle$ **do**
14:     lineSweepMining(Par_t)
15: **end for**

As shown in Algorithm 1, the TRM algorithm contains three steps. First, in the map phase, each snapshot is keyed with its timestamp (lines 1-6). Second, in the partition phase, every snapshot is grouped with its next $(\lceil \frac{K}{L} \rceil - 1) * G + 2K$ snapshots to form a partition (lines 7-11). We will shortly discuss how the group size is derived. Third, in the reduce phase, a lineSweepMining method is invoked to mine GCMP within each partition (lines 12-14). It is easy to see that this method replicates a snapshots at most $(\lceil \frac{K}{L} \rceil - 1) * G + 2K$ times.

5.0.1 Temporal Replication Partition

The size of replication is critical for the performance of TRM algorithm. If the size of replication is too large, the shuffle cost as well as the reduce cost would be high. On the contrary, if the size of replication is too small, the *completeness* and *soundness* properties cannot be satisfied. In the Algorithm 1, the partition size is chosen as $(\lceil \frac{K}{L} \rceil - 1)*G+2K$. As stated in the following theorem, such a partition method is sound and complete.

Theorem 1 (Soundness and Completeness of Replication)**.** *Let $\mathbb{P}$ be as follows: for each snapshot S_t, create a partition $Par_t = \{S_t, ..., S_{t+(\lceil \frac{K}{L} \rceil -1)*G+2K}\}$. Then $\mathbb{P}$ is sound and complete.*

Proof. The soundness of partition can be observed from the fact that each partition represents partial trajectories with consecutive snapshots, therefore patterns in a partition can be directly mapped back to original trajectories. Given a valid pattern P, let $T' \subseteq P.T$ be the subsequence of $P.T$ which conforms to K, L, G with the smallest size. Note that there could be many qualified T's. Let the i^{th} local-consecutive part of T' be l_i and let the i^{th} gap of T' be g_i. Then, the size of T' can be written as $\Sigma_i(l_i + g_i)$. Since T' conforms to K, L, G, then $2K \geq \Sigma_i(l_i) \geq K$, $l_i \geq L$, $g_i \leq G$. Therefore, $\Sigma_i(l_i + g_i) \leq (\lceil \frac{K}{L} \rceil - 1) * G + 2K$. Thus ensuring each Par_t to be of that size would capture at least one of the T's, therefore the pattern P would be valid in Par_t. This proves the completeness of the partitioning method. ☐

5.0.2 Line Sweep Mining

After partition, each task in the reduce phase processes a partition Par_i, which contains $(\lceil \frac{K}{L} \rceil - 1)*G+2K$ snapshots starting from snapshot S_i. With such a partition method, we observe that within Par_i, only the patterns whose object sets are contained in the first snapshot are necessary to be reported. Therefore, we design a simple *line-sweep mining*(LSM) method for discover GCMPs. The algorithm works as in Algorithm 2.

The algorithm scans the snapshots in a partition in sequential order. During the scan, it maintains a candidate set C which could potentially be a valid pattern (line 1). The algorithm starts by inserting clusters at S_1 to C (lines 2-4). Subsequently, in each iteration, clusters in C are joined with clusters at S_i to generate a new set of patterns N(lines 6). Any valid new patterns are inserted back to C and any invalid patterns are discarded(lines 8 and 11).

Example 1. *We illustrate the process of TRM in Figure 5 using $M = 2, K = 2, L = 2, G = 2$. In (a), snapshots*

Algorithm 2 Line Sweep Mining

```
Require: Par_t = {S_t, S_{t+1}, ...}
C ← {}                                        ▷ Candidate set
for c ∈ S_t do
    C.add(⟨c, t⟩)
end for
for i = 1; i < |Par_t|; i++ do
    N ← S_i ⊕ C
    for all n ∈ N do
        if |n.O| ≥ M then C.add(n).
        end if
    end for
    remove unqualified candidate from C
end for
output qualified candidate in C
```

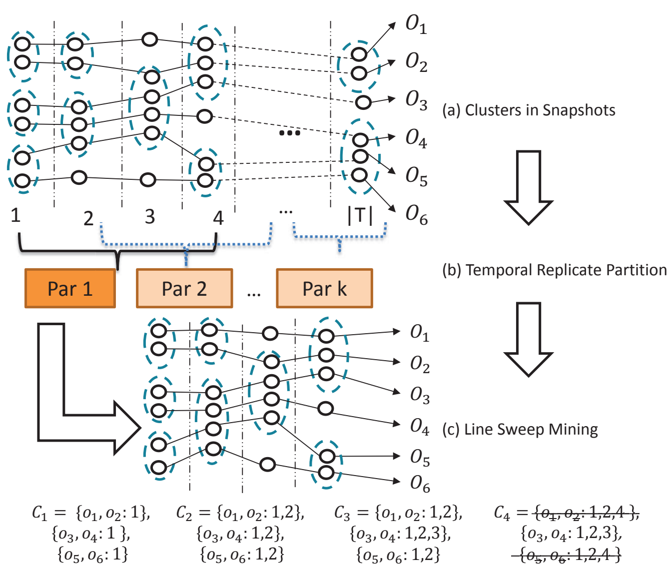


Figure 5: Work flow of trajectory replication and mining

are clustered and these snapshots are the input to the TRM. Then, we compute the size for each partition, which equals to $(\lceil \frac{K}{L} \rceil - 1) * G + 2K = 4$*. Therefore, in (b), every four snapshots are grouped into a partition. Then a line sweep method is performed in (c) for partition 1. Each* C_i *refers to the candidate set when the algorithm sweeps each snapshot. Initially,* C_1 *contains patterns whose object set is in* S_1*. When scanning the snapshots, patterns in* C_1 *grow their timestamps. At* S_4*, since the timestamps of* $\{o_1, o_2\}$ *and* $\{o_5, o_6\}$ *are both* $\{1, 2, 4\}$ *which is neither a qualified set of timestamps nor matches G constraint, thus the two candidates are removed from* C_4*. After all snapshots are scanned, only* $\{o_3, o_4\}$ *is the qualified pattern and is outputted.*

The TRM approach though achieves good parallelism, it requires to replicate the data multiple times. Specifically, each snapshots are copied $(\lceil \frac{K}{L} \rceil - 1) * G + 2K$ times. In the cases of *swarm*, *group* and *platoon*, G is as large as $|T|$. Handling those cases is equivalent to replicate the entire snapshots to each partition, which surrenders the benefit of parallelism.

6. STAR PARTITION AND MINING

In order to achieve a good parallelism under any pattern parameters, we propose the *Star Partition and Mining* (SPM) method. In SPM, we first design a novel object-based partition method named *star partition*. Then, we adapt an *Apriori*-like method to mine GCMP patterns out of each partition. The overview of the SPM method is presented in Algorithm 3. As shown, the SPM method takes three phases. In the map phase, objects from the same cluster forms object-object pairs. The object-object pairs are then paired up with the timestamp of the snapshot to form a triplet(lines 1-8). In the partition phase, triplets with the same leading object form a *star* which will be explained shortly (lines 9-12). Lastly in the reduce phase, patterns are mined from each star structure (lines 13-16).

Algorithm 3 Star Partition and Mining

```
Require: list of ⟨t, S_t⟩ pairs
 1: —Map phase—
 2: for all C ∈ S_t do
 3:     for all (o_1, o_2) ∈ C × C do
 4:         if o_1 < o_2 then
 5:             emit a ⟨o_1, o_2, {t}⟩ triplet
 6:         end if
 7:     end for
 8: end for
 9: —Partition and Shuffle phase—
10: for all ⟨o_1, o_2, {t}⟩ triplets do
11:     group-by o_1, emit ⟨o_1, Sr_{o_1}⟩
12: end for
13: —Reduce phase—
14: for all ⟨o, Sr_o⟩ do
15:     Apriori(Sr_o)
16: end for
```

6.0.3 *Star Partition*

The intuition of the star partition is that, if two objects are part of the same pattern, they must belong to the same cluster at some snapshots. Therefore, we may link objects that belong to the same cluster to form a *connection graph*. Objects that are not connected surely fail to form a pattern. We may then partition the connection graph based on vertex connectivity s.t. mining GCMPs can be done in parallel. We define the *connection graph* and *star* as follows:

Definition 5 (Connection Graph and Star)**.** *A connection graph is an undirected graph* $G = (V : E)$*, where each* $v \in V$ *represents an object. An edge* $e(s, t) = ET \in E$ *contains all the timestamps at which* s, t *are in the same cluster, i.e.,* $\forall t \in ET, C_t(s) = C_t(t)$*. A star of a vertex* s*, denoted as* Sr_s*, is the set of incidental edges on* s *whose another ending vertex is greater than* s*. I.e,* $\forall e(s, t) \in Sr_s$*,* $s < t$*. We name* s *as the* central vertex *of* Sr_s*.*

It is notable that we require vertices in a star to be greater than its central vertex. This effectively avoids replicating edges. *Connection graph* and *star* examples are shown in Figure 6 (a) and (b). In (a), a connection graph is formed based on the example in Figure 1. In (b), 5 stars are presented. It is easy to see that, by requiring the center vertex to be the smallest vertex in a star, there are no edges been replicated. In implementation, as stated in Algorithm 3 line 4, the comparison between vertices/objects are based on the vertex/object ID.

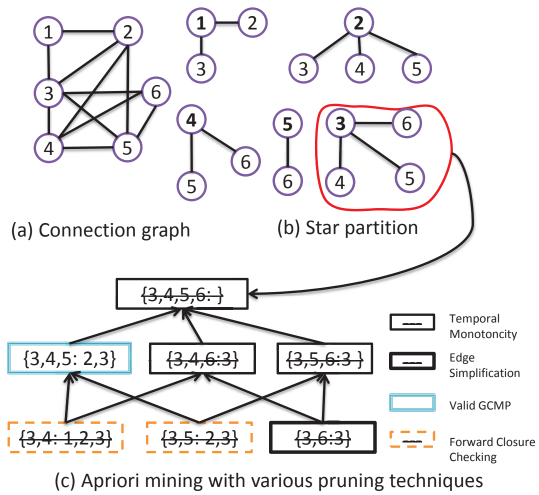


Figure 6: Star partition and mining of trajectories in Figure 1

Although star partition is performed based on the object connections, each star can be effectively viewed as a subset of trajectories. To see this, each vertex in a star can be viewed as an object. The timestamps of center vertex s is the union of all the edges in Sr_s. The timestamps of vertex $v \neq s$ is the edge $e(s, v)$. Therefore, we are able to define and mine GCMP on the stars. Before describing the mining strategy, we first state in the following theorem that the star-partition is complete and sound:

Theorem 2 (Soundness and Completeness of Star Partition)**.** *Star partition is sound and complete.*

Proof. For the soundness, if P is a valid pattern in Sr_s, then at every time t, $\forall o_1, o_2 \in P.O$, $C_t(o_1) = C_t(o_2)$. By definition, P is valid in the original trajectories. For the completeness, if P is a valid pattern in original trajectories, let s be the object with smallest ID in $P.O$. Then by the definition of pattern, $\forall t \in P.T$, $\forall o \in P.O$, $C_t(s) = C_t(o)$. It follows that all object $o \in P.O$ are in Sr_s. Furthermore, every timestamp in $P.T$ is included in Sr_s. Therefore, P is a valid pattern in Sr_s. □

Based on the above theorem, we can mine GCMP from each partition independently. It is notable that, in star partition, original data is replicated for $O(|\mathbb{O}|)$ times as each object may be sent to $O(|\mathbb{O}|)$ partitions. Since this complexity is free from pattern parameters, the star partition is more scalable than the temporal replication. In later sections, we will describe several engineering level optimization to further reduce the amount of replicated data.

6.0.4 Apriori Mining

In the mining phase, we need to find the patterns within each star. To systematically discover the patterns, we design the *Apriori Mining* method which is similar to the technique in frequent item mining literature. During the algorithm, we call a candidate pattern R-pattern if the size of its object set is R. Our algorithm runs in iterations. During each iteration R, we try to generate all $(R + 1)$-patterns. In iteration 1, the 2-pattern is the edges in Sr_s. In particular, for each $e(s, v) = ET$, pattern $p = (\{s, v\}, ET)$ is formed. During each iteration, we generate $(R + 1)$-patterns by joining R-patterns with 2-patterns. Specifically, the join between $p_1 = (O_1 : T_1)$ and $p_2 = (O_2 : T_2)$ would generate a new pattern $p_3 = (O_1 \cup O_2 : T_1 \cap T_2)$. Notice that in Sr_s, each R-pattern consists of the object s, thus the join will grow a R-pattern at most to a $(R + 1)$-pattern. Our mining algorithm stops where no further patterns are generated. The algorithm is illustrated as in Algorithm 4.

Algorithm 4 Apriori Mining

Require: Sr_s

Lv $\leftarrow \{\}$,Ground $\leftarrow \{\}$, Output $\leftarrow \{\}$
for all $e(s, t) = T \in Sr_s$ **do**
  Ground.add($\langle \{s, t\}, T \rangle$);
  Lv $\leftarrow$ Ground;
end for
while true **do**
  if Lv is not empty **then**
    LvCand $\leftarrow \{\}$
    for all $cand_v \in Lv$ **do**
      for all $cand$ $\in$Ground **do**
        $p \leftarrow cand_v$ join $cand$
        if $p.T$ is a candidate sequence **then**
          LvCand.add(p)
        end if
      end for
    end for
    if Lv is a pattern **then**
      Output.add(Lv)
      break
    end if
    Lv $\leftarrow$ LvCand
  else
    break
  end if
end while
output.addAll(Lv)
return output

An illustration of Algorithm 4 is shown in Figure 6 (c). As shown, the star $Sr_3 = \{3, 4, 5, 6\}$ initially generate three 2-candidates. At every iteration, higher level candidates are generated by joining lower level candidates. When no more candidates can be generated, the algorithm stops by outputting the valid patterns.

It is notable that Algorithm 4 takes exponential complexity to mine GCMP. There are two major factors dragging down the performance. First, the size of Sr_s affects the initial size of 2-patterns. Second, the candidates generated in each level affects the join performance. In later sections, we exploit the property of GCMP to reduce the two factors.

7. OPTIMIZATION

In this section, we describe several optimizations to the star-partition and mining algorithm. In addition, we also address some practical issues when deploying the SPM algorithm to real MapReduce based systems.

7.1 Edge Simplification

Each edge $e(s, v)$ in Sr_s contains a time sequence ET which represents the co-occurrence of s and v. We notice that the edge between s and t is not always necessary. For example, if an edge has a cardinality less than K, it is unnecessary to include this edge to Sr_s since it cannot contribute to any patterns. This motivates us to simplify the edges in Sr_s to boost the overall performance.

Our goal of edge simplification is to, given a time sequence T, find a subsequence of $T' \subseteq T$, such that T' is potentially conforms to K, L, G. And we wish $|T'|$ to be as small as possible. We star-off by observing that for every time sequence T, T can be divided into a set of maximally G-connected subsequences. Note that a maximally G-connected subsequence can potentially contribute to a pattern if it conforms to K, L. Therefore, we are able to reduce T to its maximally G-connected subseuqnces which conform to K, L.

To formally describe the idea, we define the *candidate sequence* as follows:

Definition 6 (Candidate Sequence)**.** *Given the pattern parameters: L, K, G, a sequence T is a* Candidate Sequence *if for any of its maximal G-connected sequence T', T' conforms to L, K.*

For example, let $L = 2, K = 4, G = 2$, sequence $T_1 = (1, 2, 4, 5, 6, 9, 10, 11, 13)$ is not a fully candidate sequence since one of its maximal G-connected sequence $(9, 10, 11)$ is not a partly candidate sequence. In contrast, sequence $T_2 = (1, 2, 4, 5, 6)$ is a fully candidate sequence.

To reduce a sequence T to a candidate sequence, we need to strip out its maximal G-connected subsequences which does not form to K, L. Such a reduction takes two rounds scan of T as shown in Algorithm 5. In the first round, the consecutive portions of T with size less than L are removed. In the second round, the maximal G-connected sequences of size less than K are removed. Clearly the simplification algorithm runs in linear time.

Example 2. *Take $T_1 = \{1, 2, 4, 5, 6, 9, 10, 11, 13\}$ as an example of edge simplification. Let $L = 2, K = 4, G = 2$. In the first round of scan. T_1 reduces to $\{1, 2, 4, 5, 6, 9, 10, 11\}$. The consecutive subsequence $\{13\}$ is removed by $L = 2$. T_1 has two maximal G-consecutive subsequences, which are $\{1, 2, 4, 5, 6\}$ and $\{9, 10, 11\}$. Since $K = 4$, $\{9, 10, 11\}$ is removed from T_1 in the second round of scan. Therefore, T_1 is simplified to $\{1, 2, 4, 5, 6\}$.*

By leveraging the edge simplification technique, the size of the edges in Sr_s can be greatly reduced. If an edge cannot be reduced to a candidate sequence, then it is directly removed from Sr_s. If an edge can be reduced to a candidate sequence, replacing itself by the candidate sequence results in a more compact storage.

Algorithm 5 Edge Simplification

```
Require: T
—Remove the consecutive segment with size less than L—
c ← 0
for i ∈ (0, ..., |T|) do
    if T[i] − T[i − 1]! = 1 then
        if i − c < L then
            T remove [c : i)
        end if
        c ← i
    end if
end for
—Remove the G-connected segment with size less than K—
s ← 1, c ← 0
for i ∈ (0 : |T|) do
    if T[i] − T[i − 1] > G then
        if s < K then
            T remove [c : i)
        end if
        c ← i, s ← 1
    else
        s + +
    end if
end for
```

7.2 Candidate Pruning

7.2.1 Temporal monotonicity

During the apriori phase, we repeatedly join candidate patterns in different levels to generate a larger set of a patterns. We observe that traditional monotonic property of Apriori algorithms **does not** hold in GCMP mining. That is given two candidate P_1, P_2, if $P_1.O \subset P_2.O$ and P_1 is not a valid pattern, then P_2 may or may not be a valid pattern. However, we notice that we may form another monotonic property based on the *candidate sequence* such that the Apriori algorithm could still benefit.

The intuition is that if a candidate $P_1.T$ cannot be reduce to a *candidate sequence*, then P_1 cannot be valid pattern. Furthermore, any candidate P_2, with $P_1.O \subset P_2.O$ cannot be a valid pattern. This *temporal monotonic property* is explicitly described as in the follow theorem:

Theorem 3 (Temporal Monotonic Property of GCMP)**.** *Given the temporal parameters L, G, K, for a candidate c in Algorithm 4, if $c.T$ cannot be reduced to a candidate sequence, then for any candidate c' with $c.O \subset c'.O$, c' can be pruned.*

Proof. Let c_1, c_2 be two candidates with $c_1.O \subset c_2.O$. It is easy to see that $c_1.T \supseteq c_2.T$. If $c_1.T$ cannot be reduced to a candidate sequence, then any subset of $c_1.T$ cannot be reduced. It follows that $c_2.T$ cannot be reduced neither. Thus, if $c_1.T$ cannot be reduced to a candidate sequence, c_2 can be pruned. □

7.2.2 Forward closure checking

Although leveraging *temporal monotonicity* could largely prune false candidates and reduce the apriori search space, it is ineffective when a *true* pattern exists. For example, if a final pattern of a star Sr_s is the *union of all vertices* in the star, then in apriori, $\binom{|Sr_s|}{i+1}$ candidates needs to be generated at each level i. This results in an exponential search space while the output only contains one pattern. In general, when candidates at level i collectively forms a true pattern, running aprior produces many wasted candidates.

Let Lv_i be the set of candidates at level i of Algorithm 4, we use the *forward closure FC_i* to denote the union of the objects in all candidates in Lv_i. Then, the *forward closure checking* is stated as follows:

Theorem 4 (Forward Closure Checking Rule)**.** *Let Lv_i be the candidates generated at level i in Algorithm 4, if FC_i*

is a proper pattern, then it is safe to terminate Algorithm 4 and directly output FC_i.

Proof. We prove by contradiction. Suppose there exists another pattern P such that $P.O \neq FC_i$, let $X = P.O - FC_i \neq \emptyset$. Consider a subset of P which contains X with size $i+1$, (i.e., $P_1 \subseteq P, P_1 \subseteq X, |P_1| = i+1$). Since P is a proper pattern, then P_1 is also a proper pattern. Therefore $P_1 \in Lv_i$. Then it follows X is in the forward closure of FC_i, (i.e.,$X \in FC_i$), which contradicts with $X \notin FC_i$. □

It is notable that, as the level grows in Algorithm 4, the closure FC reduces, thus the pruning power of FC would be stronger.

Example 3. *We use Figure 6 (c) to demonstrate the power of candidate pruning. As shown, at the initial stage, $\{3, 6 : 3\}$ is first pruned by Edge Simplification since its timestamps fails to be a candidate sequence. Subsequently, all further candidates containing $\{3, 6\}$ are pruned by Temporal Monotonicity. Then, we check the Forward Closure of remaining candidates (i.e., $\{3, 4\}$ and $\{3, 5\}$) and find $\{3, 4, 5\}$ is a valid candidate. Therefore, $\{3, 4\}$ and $\{3, 5\}$ are pruned, and $\{3, 4, 5\}$ is the output.*

7.3 Optimal Star Partition and Load Balance

As we see in Section 6, the apriori phase in SPM may take exponential time. An important factor affecting the performance of SPM is the size of stars. We notice that, the size of star is related to the way the vertexes are numbered.

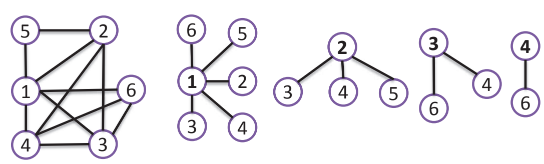


Figure 7: An alternative numbering and partitioning of the connection graph in Figure 6.

For example, Figure 7 gives an alternative numbering of vertexes in connection graph thus produces a different set of stars. This partitioning constructs four stars with the maximum star consisting of 5 edges. Compared to the partitioning in Figure 6(b) where five stars are produced with maximum star of 3 edges, this partition is inferior in two aspects. First, this partition lacks of one star, which results in a smaller number of parallelism. Second, this partition has a larger size of maximum edges. With the exponential time complexity in Apriori, the largest star takes much more time to compute.

The example shows that different numbering of vertexes in connection graph indeed affects the SPM performance. Therefore we wish to find a numbering scheme of connection graph such that the edges in each star are best distributed. Notice that the sum of edges is invariant for any numbering scheme of the graph. Thus, our goal is equivalent to find a numbering such that the maximum edge size from all stars is minimized.

To formalize the objective, we design an linear algebra model as follows: Let G denote a connection graph. Let $\mathbb{A}$ be an arbitrary numbering of vertexes in G. Let $(A : a_{i,j})$ be the matrix representing the induced graph wrt. $\mathbb{A}$. Let vector $\vec{b}$ be the *one*[1] vector. Let $\vec{c} : c_j$ be the vector equals to $A\vec{b}$. It is easy to see that each c_j denotes the size of star for vertex j. Therefore, the objective of load balancing can be formalized as follows:

$$\mathbb{A} = \text{argmin}(||A\vec{b}||_\infty), \text{where } ||A\vec{b}||_\infty = \max_{1 \leq j \leq n}(c_j) \quad (1)$$

Though the objective is well-defined as above, it is challenging to directly optimize the equation. First, suppose there are n vertex in G, enumerating all possible $\mathbb{A}$s leads to $n!$ combinations. Such a high complexity is trivially unpractical. Second, since G is constructed at runtime, the load planning can only start after G is created. The planning time are thus required to be short enough otherwise the benefit cannot payoff the planning time.

Despite these challenges, we observe that there is a $O(1)$ time solution which is good enough as stated in the following theorem.

Theorem 5 (Balance of Star Partition)**.** *Let G be a connection graph with n vertexes and the average degree d. Let $\mathbb{A}^*$ be the optimal numbering wrt. Equation 1. For any numbering, $\mathbb{A}$, with high probability, the absolute difference between $\mathbb{A}^*$ and $\mathbb{A}$ is $O(\sqrt{n \log n})$. That is, it is very likely that $||\mathbb{A}\vec{b}||_\infty = ||\mathbb{A}^*\vec{b}||_\infty + O(\sqrt{n \log n})$.*

Proof. Let $\mathbb{A}^*$ be the optimal solution wrt Equation 1. Since we have a star for each object, by the degree-sum formula and pigeon-hole theorem, $||A^*\vec{b}||_\infty \geq d/2$. Next, let $e_{i,j}$ be a 0-1 indicator variable determining whether vertex i connects vertex j in G. Note that edges in G are independent. We use d_i to denote the degree of object i in G. It follows that $E[d_i] = E[\Sigma_{1 \leq j \leq n} e_{i,j}] = d$. Since in the star partition, each edge is assigned to the ending vertex with lower ID, the connection between $a_{i,j}$ and $e_{i,j}$ can be written as:

$$a_{i,j} = \begin{cases} e_{i,j}, i > j \\ 0, otherwise \end{cases}$$

There are two observations made on the above equation. First, since $e_{i,j}$s are independent, $a_{i,j}$s are independent. Second, since $i > j$ and $e_{i,j}$ are independent. $E[a_{i,j}] = E[e_{i,j}]E[i > j] = E[e_{i,j}]/2$.

By definition, $c_i = \Sigma_{1 \leq j \leq n} a_{i,j}$, is a sum of n independent 0-1 variables. Taking expectation on both sides, we get: $E[c_i] = E[\Sigma_{1 \leq j \leq n} a_{i,j}] = E[\Sigma_{1 \leq j \leq n} e_{i,j}]/2 = d/2$. Let $\mu = E[c_i] = d/2$, $t = \sqrt{n \log n}$, by Hoeffding's Inequality, the following holds:

$$\begin{aligned} Pr(c_i \geq \mu + t) &\leq \exp(\frac{-2t^2}{n}) \\ &= \exp(-2 \log n) = n^{-2} \end{aligned}$$

The first step is due to the fact that all $a_{i,j}$ are bounded in the range of [0,1]. Next, since the event $(\max_{1 \leq j \leq n}(c_j) \geq \mu + t)$ can be viewed as $\cup_{c_i}(c_i \geq \mu + t)$, by Union Bound, we achieve the following:

$$\begin{aligned} Pr(||A\vec{b}||_\infty \geq \mu + t) &= Pr(\max_{1 \leq j \leq n}(c_j) \geq \mu + t) \\ &= Pr(\cup_{c_i}(c_i \geq \mu + t)) \\ &\leq \Sigma_{1 \leq i \leq n} Pr(c_i \geq \mu + t) \\ &= n^{-1} = 1/n \end{aligned}$$

[1]Every element in $\vec{b}$ is 1

Substitute t and μ, we achieve the following concise form:

$$Pr(||A\vec{b}||_\infty \geq (d/2 + \sqrt{n \log n})) \leq 1/n$$

This indicates that, the probability of $(||A\vec{b}||_\infty - d/2)$ being less than or equal to $O(\sqrt{n \log n})$ is $(1 - 1/n)$. With the observed fact that $||A^*\vec{b}||_\infty \geq d/2$, we conclude that with probability greater than $(1 - 1/n)$, the difference between $||A\vec{b}||_\infty$ and $||A^*\vec{b}||_\infty$ is less than $O(\sqrt{n \log n})$. □

In fact, we have a tighter bound of $||A\vec{b}||_\infty - ||A^*\vec{b}||_\infty$ if the connection graph is *dense*. Specifically, if $d \geq \sqrt{12 \log n}$, the following equation holds:

$$Pr(||A\vec{b}||_\infty \geq (d/2 + O(\sqrt{d \log n})) \leq 1/n$$

Induced by the Theorem 5, taking object IDs as the numbering (i.e., in Algorithm 3) would be guaranteed to produce a partition which is almost optimal.

8. EXPERIMENTAL STUDY

8.1 Implementation Issues

We use Apache Spark [2] as the experimental platform. Spark is one of the most popular MapReduce-like platform which uses in-memory cache to gain high speedup against Apache Hadoop []. Since Spark directly supports the MapReduce paradigm, our algorithm is able to be easily implemented in Spark. In order to help reproduce our experiments, we further address some implementation issues.

8.1.1 Task Assignment

In spark, each task in reduce phase is an Apriori mining phase of a star. Although our star partition method is theoretically balanced, it is still necessary to assign equal number of tasks to each executors. Spark naturally uses hashing to partition data into tasks, where such a partitioning does not care on the tasks size. In order to fully utilize the clusters, it is important to perform a weight-aware partition. In our implementation, we collect the number of edges in each star after map phase. Afterwards, we use a simple *best-fit* strategy, where we assign stars in decreasing order with their sizes and each star is assigned to the currently least-loaded executor. HERE WE MAY HAVE ANOTHER BOUND FROM LITERATURE, BUT I DIDN'T FIND YET. The injection of load balancing strategy between map and reduce phase can be naturally implemented in Spark, where the map result can be cached and the reduce phase can be paused until when the partition strategy is ready.

8.1.2 Duplication Detection

It is notable that the patterns discovered from different tasks (stars) could be redundant due to containment relationship. For example, a pattern $\{a, b, c\}$ can be discovered from the star Sr_a, while the pattern $\{b, c\}$ can be discovered from the star Sr_b. Though in most applications, such a duplicate pattern is permitted, we offer an option to eliminate these patterns. The strategy is to broadcast each reducers output to every other reducers. This can be efficiently done via *broadcast* variables [3] in Spark. Afterwards, each reduce can check whether any resulted patterns are subsumed and thus filter those patterns. Theoretically, advanced techniques, such as Bloom Filters, can be applied to efficiently deal with the duplication detection. However, as the number of final patterns are normally quit small, we leave the exploration for those techniques to the future.

[2] http://spark.apache.org/

[3] http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables

8.1.3 Handling Overlapping Clusters

When handling patterns such as *flock* and *group*, disk-based clustering on objects are applied. Such a clustering method may result one object belonging to multiple clusters. In such a case, just keeping the timestamps in the edge of connection graph is insufficient. Instead, we extend every timestemp t to a pair $\langle t, C\rangle$, where C is the set of clusters objects belong to at time t. The only adaption we need to take the join during apriori phase. Given two timestamp set T_1 and T_2, the join result of T_1 and T_2 instead of being $\{\forall t | t \in T_1 \wedge t_\in T_2\}$, it changes to $\{\forall(t, C) | t \in T_1 \wedge t \in T_2 \wedge C = (T_1.C \cap T_2.C) \wedge C \neq \emptyset\}$. It is obvious to see the *edge simplification* and *candidate pruning* still holds under this new setting.

8.2 Experimental Setup

Our experiments run on a 9-node cluster, with Apache Yarn as the cluster manager. We use 1 node for Yarn resource manager, and use the remaining 8 nodes as executors in Spark. In the cluster, each node is uniformly equipped with a 2.2GHz quad-core CPU with 32 GB memory. Internode communication is carried by the 1Gbps Ethernet. Some critical configuration of Spark is as follows:

Parameter	Value
Java Version	1.7.0
spark.driver.memory	2GB
spark.executor.cores	2
spark.executor.instances	11
spark.executor.memory	7GB
spark.master	yarn-cluster
spark.serializer	KryoSerializer

We setup HDFS on the same cluster and all trajectory data are initially stored in the HDFS. We prepare two set of large-scaled trajectories with the following properties:

9. CONCLUSION AND FUTURE WORK

10. REFERENCES

[1] Y. Zheng, Y. Liu, J. Yuan, and X. Xie, "Urban computing with taxicabs," in *Proceedings of the 13th international conference on Ubiquitous computing*, pp. 89–98, ACM, 2011.

[2] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye, "Mining periodic behaviors for moving objects," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1099–1108, ACM, 2010.

[3] J. Bao, Y. Zheng, D. Wilkie, and M. F. Mokbel, "A survey on recommendations in locationbased social networks. submitted to," *Geoinformatica*, 2013.

[4] X. Li, *Managing moving objects and their trajectories*. PhD thesis, National University of Singapore, 2013.

[5] J. Gudmundsson and M. van Kreveld, "Computing longest duration flocks in trajectory data," in *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pp. 35–42, ACM, 2006.

[6] Y. Wang, E.-P. Lim, and S.-Y. Hwang, “Efficient mining of group patterns from user movement data,” *Data & Knowledge Engineering*, vol. 57, no. 3, pp. 240–282, 2006.

[7] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, “Discovery of convoys in trajectory databases,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, 2008.

[8] Z. Li, B. Ding, J. Han, and R. Kays, “Swarm: Mining relaxed temporal moving object clusters,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 723–734, 2010.

[9] Y. Li, J. Bailey, and L. Kulik, “Efficient mining of platoon patterns in trajectory databases,” *Data & Knowledge Engineering*, 2015.

[10] J. Gudmundsson, M. van Kreveld, and B. Speckmann, “Efficient detection of motion patterns in spatio-temporal data sets,” in *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, pp. 250–257, ACM, 2004.

[11] Y. Zheng, “Trajectory data mining: an overview,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, p. 29, 2015.

[12] D. Birant and A. Kut, “St-dbscan: An algorithm for clustering spatial–temporal data,” *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 208–221, 2007.

[13] P. Laube, M. van Kreveld, and S. Imfeld, “Finding remodetecting relative motion patterns in geospatial lifelines,” in *Developments in spatial data handling*, pp. 201–215, Springer, 2005.

[14] D. H. Douglas and T. K. Peucker, “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,” *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973. doi:10.3138/FM57-6770-U75U-7727.

[15] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.